Trove: The PVFS2 Storage Interface

PVFS Development Team

October 11, 2026

1 Motivation and Goals

The Trove storage interface will be the lowest level interface used by the PVFS server for storing both file data and metadata. It will be used by individual servers (and servers only) to keep track of locally stored information. There are several goals and ideas that we should keep in mind when discussing this interface:

- **Multiple storage instances**: This interface is intended to hide the use of multiple storage instances for storage of data. This data can be roughly categorized into two types, bytestream and keyval spaces, which are described in further detail below.
- **Contiguous and noncontiguous data access**: The first cut of this interface will probably only handle contiguous data access. However, we would like to also support some form of noncontiguous access. We think that this will be done through list I/O type operations, as we don't necessarily want anything more complicated at this level.
- **Metadata storage**: This interface will be used as a building block for storing metadata in addition to file data. This includes extended metadata.
- **Nonblocking semantics**: This interface will be completely nonblocking for both file data and metadata operations. The usual argument for scalability and flexible interaction with other I/O devices applies here. We should try to provide this functionality without sacrificing latency if possible. *The interface will not require interface calls to be made in order for progress to occur.* This implies that threads will be used underneath where necessary.
- **Compatibility with flows**: Flows will almost certainly be built on top of this interface. Both the default BMI flow implementation and custom implementations should be able to use this interface.
- **Consistency semantics**: If we are going to support consistency, locking, etc, then we need to be able to enforce consistency semantics at the storage interface level. The interface will provide the option for serializing access to a dataspace and a vtag interface.
- **Error recovery**: The system must detect and report errors occurring while accessing data storage. The system may or may not implement redundancy, journaling, etc. for recovering from errors resulting in data loss.

Our first cut implementation of this interface will have the following restrictions:

- only one type of storage for bytestreams and one type for keyvals will be supported

- consistency semantics will not be implemented
- errors will be reported, but no measures will be taken to recover
- noncontiguous access will not be enabled
- only one process/thread will be accessing a given storage instance through this interface at a time

PARTIAL COMPLETION SEMANTICS NEED MUCH WORK!!!

2 Storage space concepts

A server controls one storage space.

Within this storage space are some number of *collections*, which are akin to file systems. Collections serve as a mechanism for supporting multiple traditional file systems on a single server and for separating the use of various physical resources. (Collections can span multiple underlying storage devices, and hints would be used in that case to specify the device on which to place files. This concept might be used in systems that can migrate data from slow storage to faster storage as well).

Two collections will be created for each file system: one collection will support the dataspaces needed for the file system's data and metadata objects. A second collection will be created for administrative purposes. If the underlying implementation needs to perform disk i/o, for example, it can use bstream and keyval objects from the administration collection.

A collection id will be used in conjunction with other parameters in order to specify a unique entity on a server to access or modify, just as a file system ID might be used.

3 Dataspace concepts

This storage interface stores and accesses what we will call *dataspaces*. These are logical collections of data organized in one of two possible ways. The first organization for a dataspace is the traditional "byte stream". This term refers to arbitrary binary data that can be referenced using offsets and sizes. The second organization is "keyword/value" data. This term refers to information that is accessed in a simplified database-like manner. The data is indexed by way of a variable length key rather than an offset and size. Both keyword and value are arbitrary byte arrays with a length parameter (i.e. need not be readable strings). We will refer to a dataspace organized as a byte stream as a bytestream dataspace or simply a *bytestream space*, and a dataspace organized by keyword/value pairs as a keyval dataspace or *keyval space*. Each dataspace will have an identifier that is unique to its server, which we will simply call a *handle*. Physically these dataspaces may be stored in any number of ways on underlying storage.

Here are some potential uses of each type:

- Byte stream
  - traditional file data
  - binary metadata storage (as is currently done in PVFS 1)
- Key/value

- extended metadata attributes
- directory entries

In our design thus far (reference the system interface documents) we have defined four types of *system level objects*. These are data files, metadata files, directories, and symlinks. All four of these will be implemented using a combination of bytestream and/or keyval dataspaces. At the storage interface level there is no real distinction between different types of system level objects.

4 Vtag concepts

Vtags are a capability that can be used to implement atomic updates in shared storage systems. In this case they can be used to implement atomic access to a set of shared storage devices through the storage interface. To clarify, these would be of particular use when multiple threads are using the storage interface to access local storage or when multiple servers are accessing shared storage devices such as a MySQL database or SAN storage.

This section can be skipped if you are not interested in consistency semantics. Vtags will probably not be implemented in the first cut anyway.

4.1 Phil's poor explanation

Vtags are an approach to ensuring consistency for multiple readers and writers that avoids the use of locks and their associated problems within a distributed environment. These problems include complexity, poor performance in the general case, and awkward error recovery.

A vtag fundamentally provides a version number for any region of a byte stream or any individual key/value pair. This allows the implementation of an optimistic approach to consistency. Take the example of a read-modify-write operation. The caller first reads a data region, obtaining a version tag in the process. It then modifies it's own copy of the data. When it writes the data back, it gives the vtag back to the storage interface. The storage interface compares the given vtag against the current vtag for the region. If the vtags match, it indicates that the data has not been modified since it was read by the caller, and the operation succeeds. If the vtags do not match, then the operation fails and the caller must retry the operation.

This is an optimistic approach in that the caller always assumes that the region has not been modified.

Many different locking primitives can be built upon the vtag concept...

4.2 Use of vtags

Layers above trove can take advantage of vtags as a way to simplify the enforcement of consistency semantics (rather than keeping complicated lists of concurrent operations, simply use the vtag facility to ensure that operations occur atomically). Alternatively they could be used to handle the case of trove resources shared by multiple upper layers. Finally they might be used in conjunction with higher level consistency control in some complimentary fashion (dunno yet...).

Table 1: Error values for storage interface

Value	Meaning
TROVE_ENOENT	no such dataspace
TROVE_EIO	I/O error
TROVE_ENOSPC	no space on storage device
TROVE_EVTAG	vtag didn't match
TROVE_ENOMEM	unable to allocate memory for operation
TROVE_EINVAL	invalid input parameter

5 The storage interface

In this section we describe all the functions that make up the storage interface. The storage interface functions can be divided into four categories: dataspace management functions, bytestream access functions, keyval access functions, and completion test functions. The access functions can be further subdivided into contiguous and noncontiguous access capabilities.

First we describe the return values and error values for the interface. Then we describe special vtag values and the implementation of keys. Next we describe the dataspace management functions. Next we describe the contiguous and noncontiguous dataspace access functions. Finally we cover the completion test functions.

5.1 Return values

Unless otherwise noted, all functions return an integer with three possible values:

- 0: Success. If the operation was nonblocking, then this return value indicates the caller must test for completion later.
- 1: Success with immediate completion. No later testing is required, and no handle is returned for use in testing.
- -errno: Failure. The error code is encoded in the negative return value.

5.2 Error values

Table 1 shows values. All values will be returned as integers in the native format (size and byte order).

Needs to be fleshed out. Need to pick a reasonable prefix.

Phil: Once this is fleshed out, can we apply the same sort of scheme to BMI? BMI doesn't have a particularly informative error reporting mechanism.

Rob: Definitely. I would really like to make sure that in addition to getting error values back, the error values actually make sense :). This was (and still is in some cases) a real problem for PVFS1.

5.3 Flags related to vtags

As mentioned earlier, the usage of vtags is not manditory. Therefore we define two flags values that can be used to control the behavior of the calls with respect to vtags:

TODO: pick a reasonable prefix for our flags.

- **FLAG_VTAG**: Indicates that the vtag is valid. The caller does not have a valid vtag for input, nor does he desire a valid vtag in response.
- **FLAG_VTAG_RETURN**: Indicates that the caller wishes to obtain a vtag from the operation. However, the caller does not wish to use a vtag for input.

By default calls ignore vtag values on input and do not create vtag values for output.

5.4 Implementation of keys, values, and hints

TODO: sync. with code on data_sz element.

```
struct TROVE_keyval {
    void *  buffer;
    int32_t buffer_sz;
    int32_t data_sz;
};
typedef struct TROVE_keyval TROVE_keyval_s;
```

Keys, values, and hints are all implemented with the same TROVE_keyval structure (do we want a different name?), shown above. Keys and values used in keyval spaces are arbitrary binary data values with an associated length.

Hint keys and values have the additional constraint of being null-terminated, readable strings. This makes them very similar to MPI_Info key/value pairs.

TODO: we should build hints out of a pair of the TROVE_keyvals. We'll call them a TROVE_hint_s in here for now.

5.5 Functions

Note: need to add valid error values for each function.

TODO: find a better format for function descriptions.

5.5.1 IDs

In this context, IDs are unique identifiers assigned to each storage interface operation. They are used as handles to test for completion of operations once they have been submitted. If an operation completes immediately, then the ID field should be ignored.

These IDs are only unique in the context of the storage interface, so upper layers may have to handle management of multiple ID spaces (if working with both a storage interface and a network interface, for instance).

The type for these IDs is TROVE_op_id.

5.5.2 User pointers

Each function allows the user to pass in a pointer value (void *). This value is returned by the test functions, and it allows for quick reference to user data structures associated with the completed operation.

To motivate, normally there is some data at the caller's level that corresponds with the trove operation. Without some help, the caller would have to map IDs for completed operations back to the caller data structures manually. By providing a parameter that the caller can pass in, they can directly reference these structures on trove operation completion.

5.5.3 Dataspace management

- **ds_create([in]coll_id, [in/out]handle, [in]bitmask, [in]type, [in/out]hint, [in]user_ptr, [out]id)**: Creates a new storage interface object. The interface will fill any any portion of the handle that is not already filled in and ensure that it is unique. For example, if the caller wants to specify the first 16 bits of the handle, it may do so by setting the appropriate bits and then specifying with the bitmask that the storage interface should not modify those bits.

  The type field can be used by the caller to assign an arbitrary integer type to the object. This may, for example, be used to distinguish between directories, symlinks, datafiles, and metadata files. The storage interface does not assign any meaning to the type value. *Do we even need this type field?*

  The hint field may be used to specify what type of underlying storage should be used for this dataspace in the case where multiple potential underlying storage methods are available.

- **ds_remove([in]handle, [in]user_ptr, [out]id)**: Removes an existing object from the system.

- **ds_verify([in]coll_id, [in]handle, [out]type, [in]user_ptr, [out]id)**: Verifies that an object exists with the specified handle. If the object does exist, then the type of the object is also returned. Useful for verifying sanity of handles provided by client.

- **ds_getattr([in]coll_id, [in]handle, [out]ds_attr, [in]user_ptr, [out]id)**: Obtains statistics about the given dataspace that aren't actually stored within the dataspace. This may include information such as number of key/value pairs, size of byte stream, access statistics, on what medium it is stored, etc.

- **ds_setattr() ???**

- **ds_hint([in]coll_id, [in]handle, [in/out]hint);**: Passes a hint to the underlying trove implementation. Used to indicate caching needs, access patterns, begin/end of use, etc.

- **ds_migrate([in]coll_id, [in]handle, [in/out]hint, [in]user_ptr, [out]id);**: Used to indicate that a dataspace should be migrated to another medium. *could this be done with just the hint call? having an id in this case is particularly useful ... so we know the operation is completed...*

5.5.4 Byte stream access

Parameters in read and write at calls are ordered similarly to pread and pwrite.

- **bstream_read_at([in]coll_id, [in]handle, [in]buffer, [in]size, [in]offset, [in]flags, [out]vtag, [in]user_ptr, [out]id)**: Reads a contiguous region from bytestream. Most of the arguments are self explanatory. The flags are not yet defined, but may include such possibilities as specifying atomic operations. The vtag returned from this function applies to the region of the byte stream defined by the requested offset and size. A flag can be passed in if the caller does not want a vtag returned. This allows the underlying implementation to avoid the overhead of calculating the value.

  The size is [in/out] in code? Figure out semantics!!!

- **bstream_write_at([in]coll_id, [in]handle, [in]buffer, [in]size, [in]offset, [in]flags, [in/out]vtag, [in]user_ptr, [out]id)**:

  Writes a contiguous region to the bytestream. Same arguments as read_bytestream, except that the vtag is an in/out parameter.

  The size is [in/out] in code? Figure out semantics!!!

- **bstream_resize([in]coll_id, [in]handle, [in]size, [in]flags, [in/out]vtag, [in]user_ptr, [out]id)**: Used to truncate or allocate storage for a bytestream. Flags are used to specify if preallocation is desired.

- **bstream_validate([in]handle, [in/out]vtag, [in]user_ptr, [out]id)**: This function may be used to check for modification of a particular bytestream.

  Flags?

5.5.5 Key/value access

An important call for keyval spaces is the iterator function. The iterator function is used to obtain all keyword/value pairs from the keyval space with a sequence of calls from the client. The iterator function returns a logical, opaque "position" value that allows a client to continue reading pairs from the keyval space where it last left off.

- **keyval_read([in]coll_id, [in]handle, [in]key, [out]val, [in]flags, [out]vtag, [in]user_ptr, [out]id)**: Reads the value corresponding to a given key. Fails if the key does not exist. A buffer is provided for the value to be placed in (the value may be an arbitrary type).

  The amount of data actually placed in the value buffer should be indicated by the data_sz element of the structure.

- **keyval_write([in]coll_id, [in]handle, [in]key, [in]val, [in]flags, [in/out]vtag, [in]user_ptr, [out]id)**: Writes out a value for a given key. If the key does not exist, it is added.

- **keyval_remove([in]coll_id, [in]handle, [in]key, [in]flags, [in/out]vtag, [in]user_ptr, [out]id)**: Removes a key/value pair from the keyval data space.

- **keyval_validate([in]coll_id, [in]handle, [in/out]vtag, [in]user_ptr, [out]id)**: Used to check for modification of a particular key/value pair.

- **keyval_iterate([in]coll_id, [in]handle, [in/out]position, [out]key_array, [out]val_array, [in/out]count, [in]flags, [in/out]vtag, [in]user_ptr, [out]id)**: Reads count keyword/value pairs from the provided logical position in the keyval space. Fails if the vtag doesn't match. The position SI_START_POSITION is used to start at the beginning, and a new position is returned allowing the caller to continue where they left off.

  keyval_iterate will always read *count* items, unless it hits the end of the keyval space (EOK). After hitting EOK, *count* will be set to the number of pairs processed. Thus, callers must compare *count* after calling and compare with the value it had before the function call: if they are different, EOK has been reached. If there are N items left in the keyspace, and keyval_iterate requests N items, there will be no indication that EOK has been reached and only after making another call will the caller know he is at EOK. The value of *position* is not meaningful after reaching EOK.

- **keyval_iterate_keys([in]coll_id, [in]handle, [in/out]position, [out]key_array, [in]count, [in]flags, [in/out]vtag, [in]user_ptr, [out]id)**: Similar to above, but only returns keys, not corresponding values. *need to fix parameters*

5.5.6 Noncontiguous (list) access

These functions are used to read noncontiguous byte stream regions or multiple key/value pairs.

How do vtags work with noncontiguous calls?

The byte stream functions will implement simple listio style noncontiguous access. Any more advanced data types should be unrolled into flat regions before reaching this interface. The process for unrolling is outside the scope of this document, but examples are available in the ROMIO code.

TODO: SEMANTICS!!!!!

TODO: how to we report partial success for listio calls?

- **bstream_read_list([in]coll_id, [in]handle, [in]mem_offset_array, [in]mem_size_array, [in]mem_count, [in]stream_offset_array, [in]stream_size_array, [in]stream_count, [in]flags, [out]vtag(?), [in]user_ptr, [out]id)**:
- **bstream_write_list([in]coll_id, [in]handle, [in]mem_offset_array, [in]mem_size_array, [in]mem_count, [in]stream_offset_array, [in]stream_size_array, [in]stream_count, [in]flags, [in/out]vtag(?), [in]user_ptr, [out]id)**:
- **keyval_read_list([in]coll_id, [in]handle, [in]key_array, [in]value_array, [in]count, [in]flags, [out]vtag, [in]user_ptr, [out]id)**:
- **keyval_write_list([in]coll_id, [in]handle, [in]key_array, [in]value_array, [in]count, [in]flags, [in/out]vtag, [in]user_ptr, [out]id)**:

5.5.7 Testing for completion

Do we need coll_ids here?

- **test([in]coll_id, [in]id, [out]count, [out]vtag, [out]user_ptr, [out]state)**: Tests for completion of a storage interface operation. The count field indicates how many operations completed (in this case either 1 or 0). If an operation completes, then the final status of the operation should be checked using the state parameter. Note the vtag output argument here; it is used to provide vtags for operations that did not complete immediately.
- **testsome([in]coll_id, [in/out]id_array, [in/out]count, [out]vtag_array, [out]user_ptr_array, [out]state_array)**: Tests for completion of one or more trove operations. The id_array lists operations to test on. A value of TROVE_OP_ID_NULL will be ignored. Count is set to the number of completed items on return.

  TODO: fix up semantics for testsome; look at MPI functions for ideas.

  wait function for testing purposes if nothing else?

Note: need to discuss completion queue, internal or external?

Phil: See pvfs2-internal email at
http://beowulf-underground.org/pipermail/pvfs2-internal/2001-October/000010.html for my thoughts on this topic.

5.5.8 Batch operations

Batch operations are used to perform a sequence of operations possibly as an atomic whole. These will be handled at a higher level.

6 Optimizations

This section lists some potential optimizations that might be applied at this layer or that are related to this layer.

6.1 Metadata Stuffing

In many file systems "inode stuffing" is used to store the data for small files in the space used to store pointers to indirect blocks. The analogous approach for PVFS2 would be to store the data for small files in the bytestream space associated with the metafile.